Author(s): Falcarin, Paolo; Alonso, Gustavo.
Article title: Software Tampering Detection using AOP and mobile code
Year of publication: 2004
Citation: Falcarin, P., Alonso, G. (2004) "Software Architecture Evolution through Dynamic AOP" In: Oquendo, F. et al. (Eds) European Workshop on Software Architectures (EWSA), Saint-Andrews (UK), May 21-22, 2004 Proceedings. LNCS 3047, Heidelberg: Springer-Verlag. pp 57-73.
Link to published version: http://dx.doi.org/10.1007/b97879
DOI: 10.1007/b97879

Software Architecture Evolution through Dynamic AOP

Paolo Falcarin[1] and Gustavo Alonso[2]

[1] Dipartimento di Automatica e Informatica, Politecnico di Torino,
I-10129 Torino, Italy
Paolo.Falcarin@polito.it
[2] Department of Computer Science, Swiss Federal Institute of Technology (ETHZ)
CH-8092 Zurich, Switzerland
Alonso@inf.ethz.ch

Abstract. Modern computing and network environments demand a high degree of adaptability from applications. At run time, an application may have to face many changes: in configuration, in protocols used, in terms of the available resources, etc. Many such changes can only be adequately addressed through dynamic evolution of the software architecture of the application. In this paper, we propose a novel approach to dynamically evolve a software architecture based on run-time aspect oriented programming. In our framework, a system designer/administrator can control the architecture of an application by dynamically inserting and removing code extensions. It is even possible to replace a significant part of the underlying middleware infrastructure without stopping the application. The novelty of this work is that it allows for a much more flexible development strategy as it delegates issues like middleware choice and adherence to an architectural specification to a framework enhanced by dynamic code extensions.

1 Introduction

Software architectures for distributed systems are a challenge in terms of software development and evolution. Design choices like, e.g., the kind of architecture, and the underlying middleware among the components are often made in an early design phase, and are therefore difficult and expensive to alter or rollback. To minimize the impact and cost of such design changes, the notion of software variability has been introduced [1]. Software variability implies a series of locations in the software where behavior and structure can be configured as well as the ability to change, customize or configure different aspects of the system. Moreover, to keep the evolution under control, variability requires a model-driven and architecture-centric approach that constraints changes and avoids undesired divergences as the specification evolves. In this paper we explore the issue of variability in the area of distributed systems. In particular, we are interested in the interplay between middleware platforms and component models, and how this aspect can be treated as a configurable option, preferably at run time. Our

goal is to address some of the challenges encountered when using some of the predominant component platforms [2]: CORBA/CCM [3], J2EE/EJB [4], or Web Services [5]. For instance, in the context of these platforms, it has been argued in favor of agile development processes [6].
Agile methods suggest a continuous process whereby working software is constantly being produced as the development progresses toward the final objectives. Yet, in many applications, particularly in the area of distributed systems, having a working prototype already implies that several crucial design decisions have been made: for example, the component model to use and, by association, the underlying middleware infrastructure. Such early design decisions limit the scope of agility because they may become too costly to revisit or readjust at a later point in time. To address these issues, we propose a mechanism to implement and specify variability at both development and run time. At development time, the idea is to support delaying architectural decisions, such as the type of middleware platform to be used, with minimal impact on reconfiguration and modifications. At run time, the former idea is extended to insert/withdraw connectors and components in the deployed architecture, without interrupting the application.
The framework we propose combines ideas from dynamic Aspect-Oriented Programming (AOP) [7] and dynamic software architectures [8]. Its contribution is to provide a mechanism whereby middleware infrastructure, components, and overall system architecture are treated as software variants that can be easily changed during prototyping or even at run time. Thus, the framework provides the necessary flexibility to adapt to continuous changes either for rapid prototyping purposes or as a result of changes in the computing or network environment.
The paper is organized as follows: first of all we introduce the motivation of our work, then we describe our framework in detail, and finally we show a case study and discuss related work.

2 Motivation and requirements

In this section we discuss the motivation behind the proposed framework and previous work the framework builds upon.

2.1 Middleware and flexible development

The early life cycle stages of specification and design are of crucial importance in a distributed system. Typically, it is at these stages that the middleware platform is chosen. Because the middleware platform tends to have such a profound effect on the architecture and properties of the resulting system, decisions related to the underlying middleware are particularly significant. For example, if language independence (or heterogeneity) is a requirement, then CORBA is a natural choice for middleware. However, once CORBA is selected, then the interfaces between components must be specified through the CORBA IDL, middleware services are component managed, and access to middleware services is through

a particular programming model whereby the services used are determined at compile time. In contrast, a Java-based system may choose to use Enterprise Java Beans. In this case the middleware services are (by default) container-managed, and access to the services is by a different programming model. The decisions about which services are used and how they are used are made not at compile time but at deployment time. These differences are significant enough to constraint the degree of freedom in the overall design and, in most realistic applications, are very costly to change once development has started. Ideally, services and even the middleware platform itself should be parameters of the system that can be changed, as need dictates. The objective would be to use a declarative specification of all design decisions that are to become variants (to simplify development and separate concerns) but making these decisions explicit by formalizing them in an architectural specification file, and by building a framework that implements it in the current architecture of the system.

2.2 Software architecture evolution

Software connectors provide a uniform interface abstraction of communication to other connectors and components of the architecture: thus, designers need not to be concerned with the properties of different middleware technologies, if the technology can be encapsulated within a software connector. Moreover, the advantages of combining multiple middleware technologies, to be used in different parts of a distributed architecture, are even more evident with separation of connector code from component code.
This separation, beyond leading to an easier development, is then a necessary condition to support dynamic evolution of software architectures, i.e. runtime reconfiguration and/or replacement of components and connectors in a running system. We need a specification language that allows to easily define and modify architectural elements, and to be executed by a framework supporting dynamic evolution.
At the implementation level, to support dynamic evolution, the programming style changes: the programmer writes application code that must be independent from middleware-specific mechanisms and thus treating each remote method call as a normal local call; moreover design decisions about technologies used by an architecture are automatically reflected in the running system by means of a framework and must not worry anymore application developer. Following these ideas, our framework will rely on an architectural specification defined with xADL (XML-based Architecture Description Language) [9]; there are several ADLs focused on dynamic software architectures, but we have chosen xADL because: it is designed to be a standard way to express architectural specification; it is extensible and adaptable allowing architects to define new XML-Schemas for extensions that can be referenced in a xADL file; moreover, it is based on XML and it is easier to be automatically parsed than other ADLs. The latter feature is a key point to enable a real mapping between code and its architectural specification, even if they both evolve in time.

2.3 Dynamic adaptation

Once certain design decisions can be postponed and have been made explicit through a document that the system uses to configure the application, it is possible to take the idea one step further. Rather than making architectural decisions at development or deployment time, they can be made also at run time. For example, it should be possible to dynamically change the middleware platform used at run-time with a new version of the same platform or an even different platform altogether. We will later demonstrate how this can be done by using the proposed framework to change a distributed application running on CORBA to a Web Service based implementation. The change can take place without stopping any system component, and they are done under the control of a specification decided by the system architect and propagated through a centralized configuration application.
This is a key issue if components are deployed on remote terminals, where manual reconfiguration is not feasible, and it is very important to maintain application integrity and coherence with the evolving specification. The use of a specification and basing all changes on the specification document allows for all necessary checks to be performed at run-time. Such checks are part of the functionality of our framework, which attaches an instance of the framework to each component to make sure the checks are performed. In addition, our framework can be coupled with a dynamic Aspect-Oriented Programming platform (the PROSE system, discussed later on in the paper) for added flexibility. Using dynamic AOP gives us the possibility to remotely insert middleware code and model-checking features of the framework in advice code. This advice code is then directly tied to the component rather than executed separately, thereby allowing us to dynamically insert and withdraw these aspects at run-time as the specification changes. Thanks to PROSE the dynamic adaptation can be efficient and do not halt normal operations, and changes can be propagated to all distributed components in a reliable and transactional manner.

3 JADDA framework

In this section we discuss the implementation of our framework: JADDA, Java Adaptive component for Dynamic Distributed Architectures. JADDA has two parts: a Java library used to check architectural specification and to handle middleware concerns, and a System Administrator Console (SAC) used to propagate the xADL architectural specification to all the distributed components using JADDA. SAC is an independent application that is used to send xADL specification files to all involved remote servers of distributed systems, while the JADDA library has to be included in all components of the distributed architecture.
In the following subsections we describe different features of JADDA implementation. First of all we describe its inner architecture and API, then we detail xADL standard extensions created to configure middleware (like CORBA and SOAP); moreover we describe extensions mechanism based on a dynamic AOP

platform, and JADDA behavior during run-time reconfiguration due to evolution of architectural specification.

3.1 JADDA architecture

JADDA library that must be included in each component of the distributed architecture has an inner structure depicted with the UML class diagram of Figure 1.

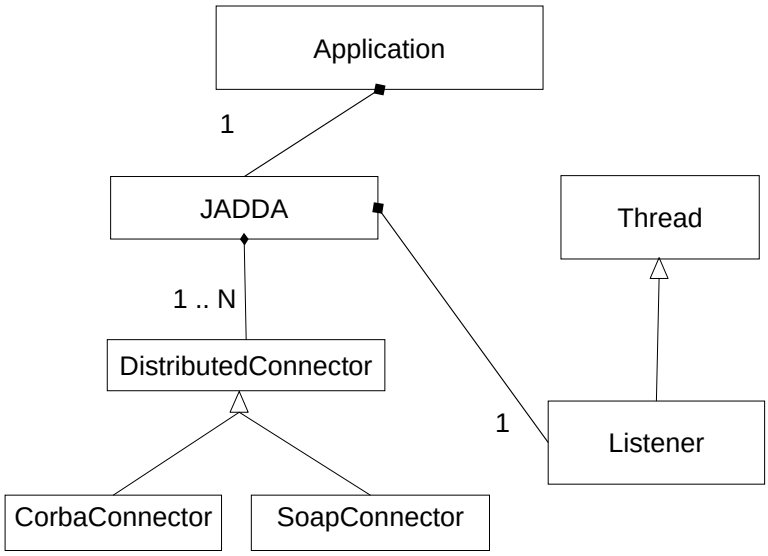

Fig. 1. JADDA UML class diagram

Each application must include an instance of JADDA class; during its construction and initialization, a Listener instance is created in a separate thread. The Listener has the complex task to listen on the network for incoming xADL specification file and to handle dynamic reconfiguration. Moreover JADDA may include different instances of DistributedConnector abstract class: this defines a common API for middleware: in fact different reifications of this class (like CorbaConnector and SoapConnector) can be added to implement behavior of different middleware standards. During initialization the JADDA instance, running in each component, registers itself in the JADDA System Administrator Console in order to receive the current xADL architectural specification file; then all the needed remote interface references are taken by the CORBA Name server or by the UDDI [10] registry depending on the information contained in the xADL file. Application independence from the middleware used is due to the fact that JADDA wraps on different middleware protocols for remote method invocation, offering to application a simple API, whose typical usage is depicted in Figure 2.

VI

```
Jadda jadda = new Jadda();

Service s =newService("Server","ChatManager");

String methodName="accessRoom";

String parameter = "Joe";

jadda.call(s, methodName, parameter);
```

Fig. 2. Remote invocation with JADDA API

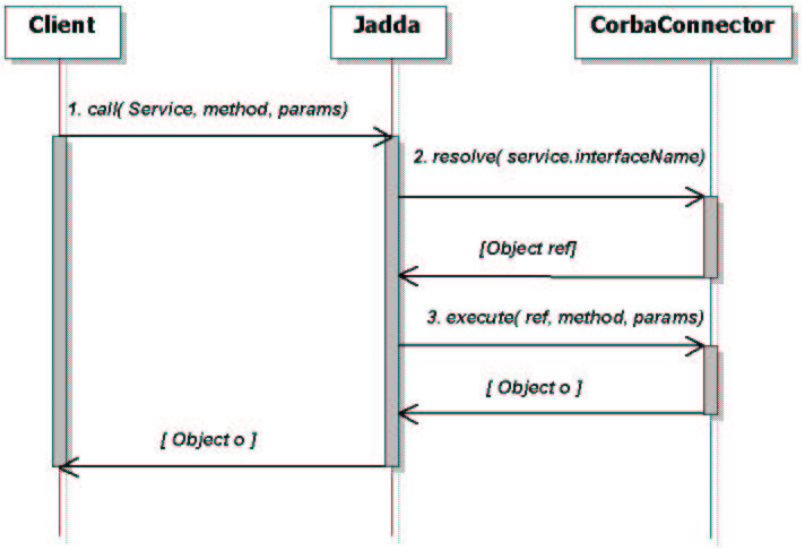

Fig. 3. Sequence diagram of method 'call'

The method "call" is overloaded in order to offer different versions able to call methods with different numbers of parameters; in the different 'call' method signatures, after the first three strings identifying the requested method, the remaining parameters are all Java 'Object' types: they all refer to the main 'call' method implementation with a third parameter made by an array of Object classes. We introduce the Service class to represent a generic remote reference to a service; looking at the previous code the creation of a Service object with the depicted parameters, creates a generic reference to the interface "ChatManager" of the component "Server". These values have to be present in the specification file, because the method 'call' searches in the current xADL file all the informa-

tion about the middleware needed to communicate with the requested interface method and it uses Java reflection to execute the remote method invocation, as depicted in figure 2, supposing the case of a CORBA call.

The method "resolve" on the underlying connector implementation is invoked to resolve and cache the remote reference for subsequent calls. The method "execute" realizes the real remote invocation using Java reflection to use middleware-related classes, depending on the used DistributedConnector's subclass (in this case CorbaConnector) and on related data defined in the xADL file. This approach gives a unique and abstract view of different middleware standards. This implementation strategy reduces significant problems in the development and maintenance of software systems and connector code is no more mixed with component code; thus service code is more portable and independent by middleware chosen in the beginning of design and service code can be easily reused or upgraded in future versions.

3.2 xADL extensions for distributed systems

JADDA provides a uniform interface on different software connectors: in this way system designers need not to be concerned with properties of different middleware technologies. The basic schema of xADL is reused to define the architecture's topology but new XML-schemas have been created to specify information related with distributed systems.

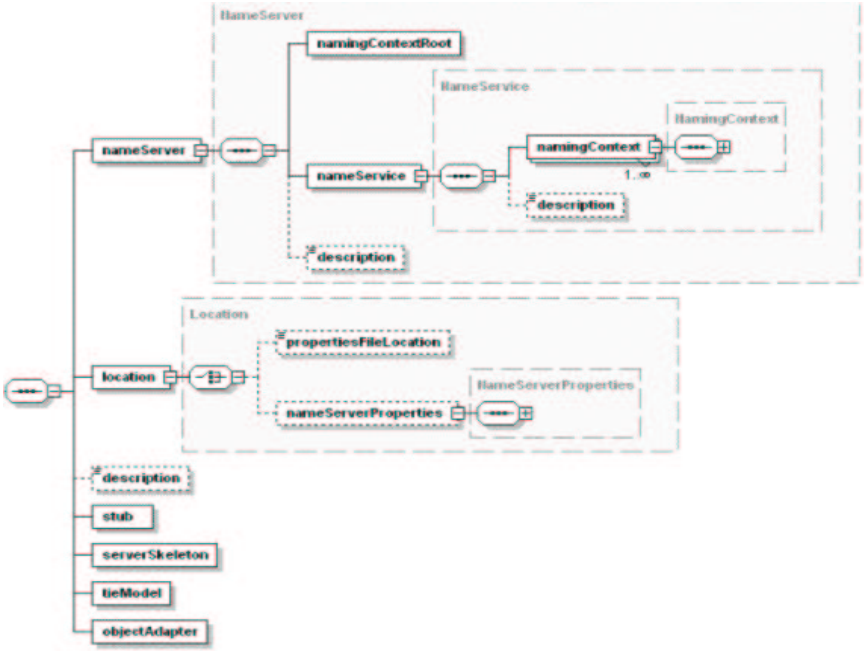

Fig. 4. CorbaConnector XML schema

For example the communication type defined in a xADL's Connector schema has been extended with a new schema called Distributed-Connector. This is only a basic schema that is specialized by other XML-schemas related to middleware protocol standards, like CorbaConnector for CORBA-IIOP, and SoapConnector for SOAP. A standard protocol like CORBA-IIOP can be implemented by different middleware platforms, offering slightly different APIs to applications. Therefore, including in the same architecture different kinds of CORBA implementations, means having different instances of CORBA-connectors in a xADL file: each CORBA connector instance will define values of tags, defined in the CORBA-connector schema, to qualify its own specializations, as sketched in figure 4.

For example, the CORBA schema defines tags like: "NameServer" that includes all the needed information for binding and retrieving CORBA object references published on a CORBA Naming Service; the tag "Location" gives the runtime information to connect to Naming Service (e.g. hostname and port). The kind of middleware used by a component's interface is defined in the extended-Interface XML-schema: this one extends the xADL's Interface schema and it contains the reference to the connector instance used by a component's interface; other XML-schemas like DistributedLink, and SoapConnector are not detailed for brevity.

3.3 JADDA reconfiguration

Once described JADDA behavior while the system is running, in this subsection we describe how architecture reconfiguration works. First of all the Listener thread included in JADDA registers its presence to the SAC, sending the IP address and port where it is waiting for a new specification file. Then SAC sends this file to all components of the distributed architecture, i.e. to all their registered JADDA instances. When the file is received the Listener thread has to follow a particular behavior, as depicted in state-chart of figure 5, passing from INIT state to REQUEST state. This Listener's state means that a new specification file has been received and that is waiting the right moment to reconfigure JADDA. In this case it checks if the main application thread containing JADDA instance is IDLE (i.e. no calls are currently in execution) or FREEZE (i.e. JADDA has terminated a remote call and it has found that Listener's state is REQUEST): if the previous conditions hold then Listener can update internal tables of JADDA in a synchronized manner, passing to the state UPDATING. Once finished it comes back to IDLE state, waiting for a new specification file from the network.

To understand the whole behavior we also have to describe the JADDA state-chart, depicted in figure 6. In the beginning and the end of each remote call, JADDA checks Listener's state. If Listener is IDLE then JADDA can execute different remote calls, passing to CALLING state and using the variable 'apps' to count the number of parallel invocations currently in execution: this is due to the fact that the main application can be multi-threaded and different parallel invocations to JADDA are possible. If Listener is in REQUEST state, JADDA

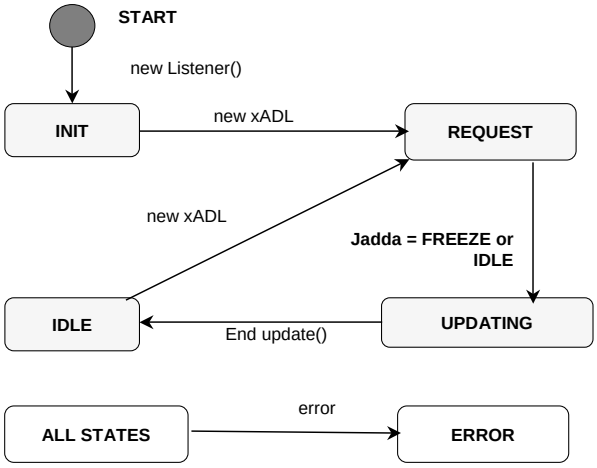

Fig. 5. Listener state-chart

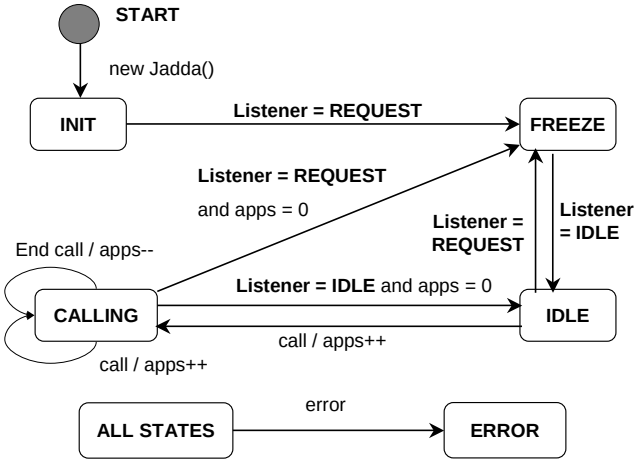

Fig. 6. JADDA state-chart

has to complete all the current remote calls (eventually suspending new incoming ones): when apps counter reaches zero then JADDA can move to FREEZE state, leaving the full control of its data to the Listener thread, that can start the reconfiguration of JADDA's internal data. The IDLE state can always be reached because of timeouts support of middleware implementations: when a re-

mote call is blocked the timeout triggers an exception that is caught by JADDA, which forces the IDLE state.

3.4 JADDA extension for dynamic AOP

Dynamic AOP is used to extend the features of an application at run time. Dynamic creation of aspects by the system designer can lead to unexpected software evolution: in our case, the scope of extensions is constrained to middleware code, i.e., future modifications of the connector implementation and configuration that were not considered in the early phases of the design. The aspects remote transmission in a transactional way allows all the components of the architecture to dynamically upload new classes. The necessary condition to obtain these results is running JADDA enabled with dynamic AOP features; moreover, when dynamic AOP is set, application developer can further run JADDA in two distinct AOP-modes: in the first one, developers can handle each remote method invocation in the code, writing local methods having the same signature as the needed methods on remote interfaces. These local methods, initially with an empty body, will be completed by the JADDA architectural framework, using the code of aspects and classes inserted at run-time by the dynamic aspect-oriented platform PROSE [11], which wraps on a standard JVM, enhancing it with dynamic AOP features.
In the second AOP-mode, developer can use JADDA as usual with its API that wraps on different connectors, but the implementation of available connectors can be updated using dynamic AOP features. PROSE is a dynamic AOP platform to be able to insert and withdraw aspects at runtime, and in our case it is used to totally decouple the application code from middleware and architectural concerns. Using this tool, the adaptability of JADDA is improved allowing the dynamic downloading of new connectors and related middleware components (and related classes, like interface stubs), when a new version or a different vendor's implementation is available for a particular middleware protocol.
In both JADDA AOP-modes aspect code is transmitted by the System Administrator Console (SAC), relying on an application of PROSE that allows remote transmission of aspects bytecode in a transactional way. In the first JADDA AOP-mode, the aspect code is built by SAC using a temporary Java file, called Aspect Template (see figure 7): it contains generic and incomplete Java code of an aspect for the PROSE platform. Depending on the values of the xADL specification file, then the SAC completes the aspect template in order to obtain different Java files, one per each interfaces' method of the whole architecture. In In figure strings here depicted in bold font, like "AspectTemplate", "CLASSNAME", and "METHODNAME" are special keywords that will be replaced by SAC with the string values defined in the current xADL specification file.
Using the example of figure 2, "CLASSNAME" will be replaced by the Chat client class name and method with the string "accessRoom". The composition logic defined in the "pointcutter" method, is interpreted by the PROSE platform that will call the "METHOD_ARGS" method (i.e. the "advice", in AOP terminology) before each execution of method "METHODNAME" of the class

```
public class AspectTemplate extends DefaultAspect
{
 Jadda jadda=Jadda.instance;
 public MethodCut crosscut = new MethodCut() {
 public void METHOD_ARGS(CLASSNAME c,Service s,REST p){
    try {
        jadda.checkParameters(s, p);
        jadda.call(s, "METHODNAME", p);
    } catch (JaddaException ex) {
    jadda.prose.exception=true;
  }
} protected PointCutter pointCutter() {
    return (Executions.before()).
            AND (Within.type("CLASSNAME")).
            AND (Within.method("METHODNAME"));
} };
```

Fig. 7. Aspect Template source code

"CLASSNAME". The Aspect name is created by SAC composing the name of the current xADL file, the class name and the method name: a new unique identifier is needed because JVM does not allow namespace conflicts, even if they are due to unloaded classes.
These Java files are aspects code for the dynamic AOP platform and they are compiled to bytecode. Finally, among the different compiled classes, SAC deduces, from the xADL specification, which are the methods invoked by each component of the distributed architecture and it sends, after the new xADL file, the new aspects to each JADDA instance. As an aspect class can refer to new classes (e.g. middleware-related interface stubs), these are also sent to the JADDA instance, through the remote transfer mechanisms offered by the dynamic AOP platform. In the second AOP-mode, SAC sends new connector implementations, depending on the current xADL specification. On the other side, when the Listener thread in the JADDA instance received a new file, it withdraws the current aspects and inserts the new ones, when they are all arrived. Then the previous state-charts are still valid in order to maintain consistency during reconfiguration, even with aspect code insertions/withdrawals.

4 Case study

We have applied JADDA to a basic example of chat system whose architecture is sketched in figure 8. A chat server publishes its own interfaces ChatManager and ChatRoom on a CORBA Naming Service; the System Administrator Console (SAC) is running and listening for requests on a specified port. Two chat clients using JADDA independently bootstrap and their own JADDA instance register their presences to the SAC and send data (e.g., the port where they

are listening for xADL file transmission). The second step is represented by the transmission of the common xADL file, containing the current architectural specification, to all the involved components, i.e. the chat clients. After that, the third step is composed by the creation of aspects code in the SAC and then the consequent transmission to the clients using the remote aspect transmission feature of PROSE. Moreover, not only aspects can be added to a running application but also other additional classes, like middleware stubs, needed to activate the distributed connector used by the clients (e.g. using a CORBA connector like in figure 8).

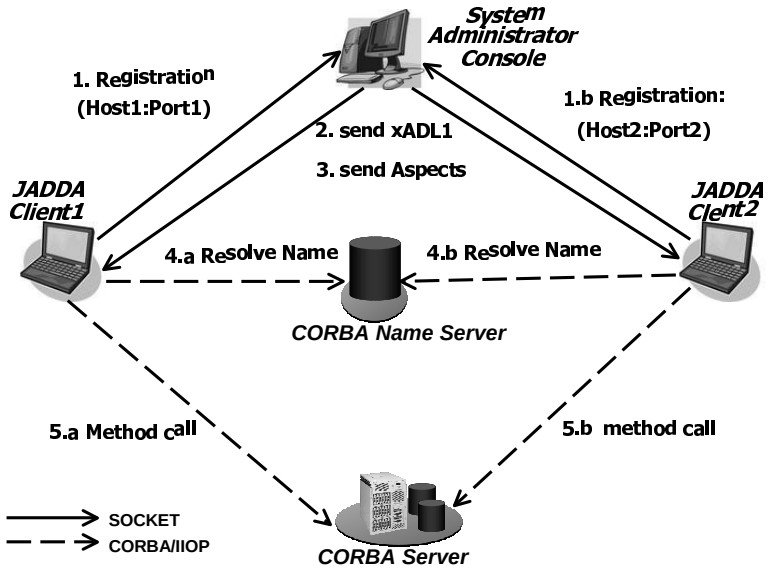

Fig. 8. Set-up scenario

Once the aspects are all arrived to a chat client, its own JADDA instance activates them and the application code starts its normal execution, resolving remote object references on the CORBA name server whom location is specified in the xADL file, and starting to call remote methods on the CORBA chat server. Let's now suppose that system architect wants to evolve the system in order to use a new middleware like SOAP and the deployed components (the chat clients) would be notified that there is a new instance of the chat server exposing WSDL interfaces on another location. System architect just needs to prepare the new xADL file with the needed information and build all the related aspects and stub classes. Once finished we can upload the new specification and the new aspects to each component currently connected using the remote aspect transmission feature of PROSE, as depicted in the first two steps of figure 9.
After all the aspects have been inserted in a chat client, it will restart a normal execution resolving remote service address through a query to the UDDI registry

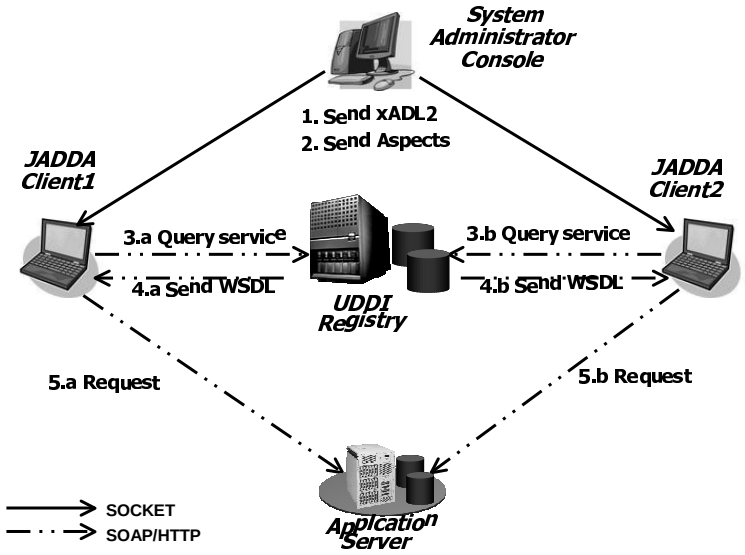

Fig. 9. Switch scenario

(step 3) and then when a WSDL interface will be sent back (step 4) a normal method call will be executed to the new deployed Chat Server. The important result is the portability of the chat application that has neither to be rewritten nor restarted while middleware is changed and components are swapped.

5 Related work

The discussion on related work touches different fields: dynamic software architectures, middleware, and separation of concerns. Architecture description languages (ADLs) and tools provide a formal basis for describing software architectures by specifying the syntax and semantics for modeling components, connectors, and configurations. Since a majority of existing ADLs, have focused on design issues, their uses have been limited to static analysis and simulate system execution at the architectural level. The other possibility, also used in our approach, is reflecting architecture modifications in an executing system. ArchJava [12] follows this approach extending Java language with new constructs and providing a compiler to build an application that adheres to its architectural specification. Different kinds of connectors are implemented, usable with an API that has some similarities with the one of JADDA, but there is no support for dynamic architectural changes. These issues are considered by tools like Regis [13], and ArchStudio [8], both made to handle architecture-based runtime software evolution. ArchStudio, like JADDA, relies on xADL architectural specification, and the run-time structure of the application is altered, generating a

different arrangement of components and connectors. These must be developed using Java-C2 class framework [14], limiting developer's freedom. While ArchStudio checks architectural properties on xADL specification, in our approach model checking is implemented at runtime in JADDA instances of each component. Neither ArchJava nor ArchStudio offer connectors relying on off-the-shelf (OTS) implementations of widespread middleware protocols (like SUN's CORBA-IIOP and SOAP in JADDA framework), but they offer its own connector implementation of other protocols.
The key role of connectors in architecture-based software development has been stated by software architecture community, and the issue of reusing OTS middleware in connectors has been recently faced [15], but, in general, existing ADLs support static description of a system, and provide no facilities for specifying both runtime architectural changes and OTS middleware encapsulation. Although a few ADLs, such as Darwin [16], C2 [14], Rapide [17], Wright [18] can express runtime modification to architectures, these are specified during design and included in the application, constraining evolution among a set of predefined alternatives. JADDA follows another approach, based on unconstrained dynamism, i.e. insertion of unexpected modifications of an architecture and incorporation of behavior not anticipated by the original developers: in this case validity of changes must be ensured both before insertion, acting on architectural model, and at runtime, preserving consistency [19]; in fact, some aspects of software architecture evolution are the same of configuration management [20], and therefore dynamic architectural changes can be seen a more generic approach to dynamically reconfigure a software system at run time. Another approach of dynamic reconfiguration is adding configuration elements to components. For instance, Polylith [21] is a specification language for configuration, used to explicitly specify component bindings: in this case reconfiguration sequence consists of two steps: waiting to reach a reconfiguration point; and blocking communication channels (managing messages in transit) during reconfiguration. A third way is delegating reconfiguration to containers [22]. JADDA dynamic reconfiguration tries to take features of the first two approaches, adding a JADDA instance to each component and governing the change with an architecture-centric approach.
Regarding middleware reconfiguration and adaptation, different approaches have been proposed for mobile applications [23] and for context-aware applications [24]. Among these ones, the architecture-centric approach was used to adapt reflective middleware to new requirements [25], using the Aster framework in supporting dynamic adaptation in the context of the Open-ORB middleware platform to accommodate re-configurations due to changing non-functional parameters and environmental conditions. This work is more focused on defining extensions of software architecture techniques to accommodate dynamic change, in order to make the automatic synthesis of component-based configurations, and their subsequent monitoring and adaptation, based entirely on architectural descriptions. It is not clear how much the architectural model is bound to the implementation as in our approach, that is also focused on the implementation of a framework able to substitute at run-time an OTS middleware implementation

(not constrained to a particular standard like CORBA) with a different one.
Moreover, several research has been done in quality-aware middleware systems [26] and middleware adaptation to non-functional features. For instance, P. Devanbu [27] proposed a methodology to enhance CORBA components with non-functional features (e.g. security), and the tool Lasagne [28] gives explicit support for CORBA clients runtime-adaptation to fulfill non-functional requirements.
This tool relies on Aspect Components [29], a dynamic aspect-oriented platform for distributed programming. It provides components that integrate system-wide properties (crosscutting concerns) such as distribution and authentication within a core application.
Separation of concerns is an emerging methodology to better modularize non-functional features, decoupled from application code. State-of-the-art separation of concerns techniques such as Aspect-oriented Programming [30], Hyperspaces [31], Mixin Layers [32], and Adaptive Plug and Play Components [33] allow extension of a core application with a new aspect/subject/layer/collaboration, by simultaneously refining state and behavior at multiple points in the application in a non-invasive manner. Therefore we think that JADDA and dynamic AOP are better suited for client-specific integration of extensions, while the above techniques are better suited for providing separation of concerns at the component implementation level. In composition filters approach [34], filters intercept messages sent and received by components. Since they are defined in extensions combined with a superimposition mechanism, they can be dynamically attached to components, but the integration of an extension is scattered across multiple object interactions, thus difficult to update consistently in one atomic action.
In JADDA, the composition logic is completely encapsulated within the composition rules contained in the aspect code (pointcuts) and the components' methods involved are taken by the specification. By doing this, developers do not need worry about architectural issues since these are automatically handled by the framework. Moreover our work extends dynamic reconfiguration to different middleware protocols and it adds basic runtime model-checking features.

6 Conclusions and future work

This paper describes the architectural framework JADDA, a component used to reach three main goals. The first one is easing timeline variability (changes that can be applied at either development time or run time) of different middleware implementations used.
The second goal is updating the connectors of a system by acting on its xADL architectural specification. This is edited and propagated by the SAC. In this way, re-configuration can be decided at a higher level than source code and all the needed information are stored in the xADL file.
Finally, dynamic reconfiguration is also handled using dynamic Aspect-Oriented Programming platform to allow additional classes transport in a reliable and transactional manner.As these features could lead to a stronger modification of application and to an unexpected software evolution, the verification of correct-

ness of a new architecture is currently implemented in JADDA to verify specified links. While at boot-time JADDA allows realization of whichever xADL architectural model in the implementation, at run-time the current JADDA implementation is tailored for simple client-server architecture, where a client application (using JADDA) interacts with only one server. For example multiple chat client applications running on mobile terminals could use JADDA to dynamically migrate their connections to a new server implementation. Current work focuses on handling more complex architectures with related consistency issues, and extending JADDA for different middleware standards. Checking xADL correctness before instantiation in the running system, and realization of different interaction styles (e.g. event-based communications) will be part of future work.

7 Acknowledgements

The authors want to thank Andrei Popovici (ETH Zurich), and Patricia Lago (Politecnico di Torino) for the precious advices given during the realization of this work.

References

1. van Gurp, J., Bosch, J., Svahnberg, M.: The notion of variability in software product lines. Proceedings 2nd Working IEEE / IFIP Conference on Software Architecture (WICSA) (2001)
2. Szyperski, C.: Component software: Beyond object-oriented programming. (1998)
3. OMG: CORBA (common object request broker architecture) specification. (http://www.corba.org/)
4. EJB: Enterprise javabeans specification. (http://java.sun.com/products/ejb/docs.html)
5. Gudgin, M., et al.: http://www.w3.org/TR/soap12-part2/. W3C Recommendation (2003)
6. Newkirk, J.: Introduction to agile processes and extreme programming. Proc. of ICSE 2002 (2002)
7. Popovici, A., Alonso, G., Gross, T.: Just in time aspects: Efficient dynamic weaving for java. Proc. of the 2nd International Conference on Aspect-Oriented Software Development (2003)
8. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. Proc. of International Conference of Software Engineering (ICSE98) (1998) 177–186
9. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: A highly-extensible, xml-based architecture description language. Proc. of Working IEEE/IFIP Conference on Software Architecture (WICSA 01) (2001) 103–112
10. UDDI: Universal discovery and description and integration. (http://www.uddi.org/)
11. PROSE: Programmable service extensions. (http://prose.ethz.ch/)
12. Aldrich, J., Chambers, C., Notkin, D.: ArchJava: Connecting software architecture to implementation. Proc. of International Conference of Software Engineering (ICSE 2002) (2002)

13. Magee, J., Dulay, N., Kramer, J.: Regis: a constructive development environment for distributed programs. Distributed Systems Engineering Journal (1994) 304–312 Special Issue on Configurable Distributed Systems.
14. Medvidovic, N., Oreizy, P., Robbins, J.E., Taylor, R.N.: Using object-oriented typing to support architectural design in the c2 style. Proceedings of the ACM SIGSOFT '96 Fourth Symposium on the Foundations of Software Engineering (1996) 24–32
15. Dashofy, E.M., Medvidovic, N., Taylor, R.N.: Using off-the-shelf middleware to implement connectors in distributed software architectures. Proc. of International Conference of Software Engineering (ICSE 99) (1999) 3–12
16. Magee, J., Kramer, J.: Dynamic structure in software architectures. Fourth SIGSOFT Symposium on the Foundations of Software Engineering (1996)
17. Luckham, D.C., Vera, J.: An event-based architecture definition language. IEEE Transactions on Software Engineering (1995)
18. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Transactions on Software Engineering (1997)
19. Feiler, P.H., Li, J.: Consistency in dynamic reconfiguration. Proc. 4th International Conference on Configurable Distributed Systems (ICCDS 98) (1998)
20. van der Hoek, A., Mikic-Rakic, M., Roshandel, R., Medvidovic, N.: Taming architectural evolution. Proc. of the Ninth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9) (2001)
21. Portilo, J.M.: The Polylith software bus. ACM Transactions on Programming Languages and Systems (TOPLAS) (1994)
22. Rutherford, M.J., Anderson, K., Carzaniga, A., Heimbigner, D., Wolf, A.L.: Reconfiguration in the Enterprise JavaBean component model. Component Deployment: IFIP/ACM Working Conference Proceedings (2002) 67–81
23. Inverardi, P., Marinelli, G., Mancinelli, F.: Adaptive applications for mobile heterogenous devices. Proc. of 22nd Intl. Conf. on Distributed Computing Systems Workshops (2002) 410–413
24. Griswold, W.G., Boyer, R., Brown, S.W., Truong, T.M.: A component architecture for an extensible, highly integrated context-aware computing infrastructure. Proc. of International Conference on Software Engineering (ICSE 2003) (2003)
25. Blair, G., Blair, L., Issarny, V., Tuma, P., Zarras, A.: The role of software architecture in constraining adaptation in component-based middleware platforms. Proc. of Middleware 2000 – IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (2000)
26. Bergmans, L., van Halteren, A., Ferreira Pires, L., van Sinderen, M., Aksit, M.: A QoS-control architecture for object middleware. Proc. of IDMS 2000 conference (2000)
27. Wohlstadter, E., Jackson, S., Devanbu, P.: DADO: enhancing middleware to support crosscutting features in distributed, heterogeneous systems. Proc. of International Conference on Software Engineering (ICSE 2003) (2003) 174 –186
28. Truyen, E., Vanhaute, B., Joosen, W., Verbaeten, P., Jorgensen, B.N.: Dynamic and selective combination of extensions in component-based applications. Proc. of the 23rd International Conference on Software Engineering (ICSE 2001) (2001) 233–242
29. Pawlak, R., Duchien, L., Florin, G., Martelli, L., Seinturier, L.: Distributed separation of concerns with aspect components. Proc. of TOOLS Europe 2000 (2000)
30. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J., Irwan, J.: Aspect-oriented programming. Proc. of ECOOP97 (1997)

31. Tarr, P., Ossher, H., Harrison, W., Sutton Jr, S.: N-degrees of separation: Multi-dimensional separation of concerns. Proc. of ICSE 99 (1999)
32. Smaragdakis, Y., Batory, D.: Implementing layered designs with mixin layers. Proc. of ECOOP 98 (1998)
33. Mezini, M., Lieberherr, K.: Adaptive plug and play components for evolutionary software development. Proc. of OOPSLA'98 (1998)
34. Aksit, M., Wakita, K., Bosch, J., Bergmans, L., Yonezawa, A.: Abstracting object-interactions using composition-filters. Object-Oriented Distributed Processing (1993) 152–184